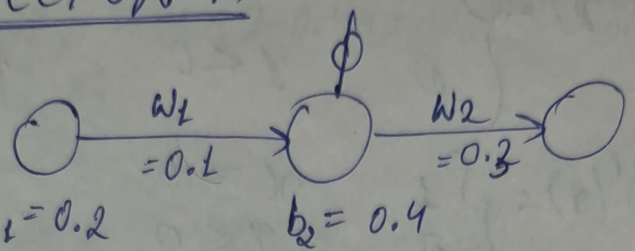
CSE343 : MACHINE LEARNING

Ritika Thakur (2022408)

THEORY : SECTION A

Q. (a)



$\phi \equiv$ ReLU

Forward pass : $y_{pred} = \phi(\underbrace{w_1 x + b_1}_{z_1}) w_2 + b_2$

Input to hidden layer : z_1

output layer : z_2

$$\text{Error} \Rightarrow \frac{1}{2}(y - y_{pred})^2$$

Backward pass :

$$\Delta w_2 = -\eta \frac{\partial E}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2}$$

$$\Delta b_2 = -\eta \frac{\partial E}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2}$$

$$\Rightarrow \Delta w_2 = \eta (y - y_{pred}) \phi(w_1 x + b_2)$$

$$\Delta b_2 = \eta (y - y_{pred})$$

$$\Delta w_1 = -\eta \frac{\partial E}{\partial z_2} \cdot \frac{\partial z_2}{\partial \phi(z_1)} \cdot \frac{\partial \phi(z_1)}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1}$$

$$\Delta b_1 = -\eta \frac{\partial E}{\partial z_2} \cdot \frac{\partial z_2}{\partial \phi(z_1)} \cdot \frac{\partial \phi(z_1)}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1}$$

$$\Delta w_1 = \eta (y - y_{pred}) \cdot w_2 \cdot \phi_1'(z_1) \cdot x$$

$$\Delta b_1 = \eta (y - y_{pred}) \cdot w_2 \cdot \phi_1'(z_1)$$

Here, $\phi(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$ $\eta = 0.01$

$\therefore \phi'(x) = \begin{cases} 1 & x \geq 0 \\ 0 & x < 0 \end{cases}$

For input sample 1,

$x = 1, \quad y = 3$

Forward pass:

$$y_{pred} = \phi(0.1x + 0.2)0.3 + 0.4$$
$$= 0.3 \times 0.3 + 0.4$$
$$= 0.09 + 0.4$$
$$= 0.49$$

$$error = \frac{1}{2}(3 - 0.49)^2 = 3.15$$

Backward pass:

$$w_{2\,new} = w_{2\,old} + \Delta w_2$$
$$= 0.3 + 0.01(3 - 0.49)$$
$$= 0.3251$$

$$b_{2\,new} = b_{2\,old} + \Delta b_2$$
$$= 0.4 + 0.01(3 - 0.49)$$
$$= 0.4251$$

$W_{1\,new} = W_{1\,old} + \Delta W_1$

$= 0.1 + 0.01\,(3 - 0.49)$

$= 0.1251$

$b_{1\,new} = b_{1\,old} + \Delta b_1$

$= 0.2 + 0.01\,(3 - 0.49)$

$= 0.2251$

For Input sample 2, $x = 2,\ y = 4$

Forward pass: $y_{pred} = \phi\,(0.125x + 0.225)\,0.325 + 0.625$

$= 0.7793$

$= $

$= $

$error = \frac{1}{2}\,(4 - 0.78)^2$

$= 5.184$

Backward pass:

$W_{2\,new} = W_{2\,old} + \Delta W_2$

$= 0.325 + 0.01\,(4 - 5.184)$

$= 0.313$

$b_{2\,new} = b_{2\,old} + \Delta b_2$

$= 0.625 + 0.01\,(4 - 5.184)$

$= 0.613$

$W_{1\,new} = W_{1\,old} + \Delta W_1$

$= 0.125 + 0.01\,(4 - 5.184)$

$= 0.113$

$b_{1\,new} = b_{1\,old} + \Delta b_1$

$= 0.225 + 0.01\,(4 - 5.184) = 0.213$

For input sample 3,

$x = 3, y = 5$

Forward pass: $y_{pred} = \phi(0.113x + 0.213)\,0.313 + 0.613$

$= 0.785$

$error = \frac{1}{2}(5 - 0.785)^2$

$= 8.88$

Backward pass:

$W_{2new} = W_{2old} + \Delta W_2$

$= 0.313 + 0.01(5 - 0.785)$

$= 0.313 + 0.042$

$= 0.355$

$b_{2new} = b_{2old} + \Delta b_2$

$= 0.613 + 0.042$

$= 0.655$

$W_{1new} = W_{1old} + \Delta W_1$

$= 0.113 + 0.042$

$= 0.155$

$b_{1new} = b_{1old} + \Delta b_1$

$= 0.213 + 0.042$

$= 0.255$

$W_1 =$

$0.1 \longrightarrow 0.155$

$W_2 =$

$0.3 \longrightarrow 0.355$

$b_1 =$

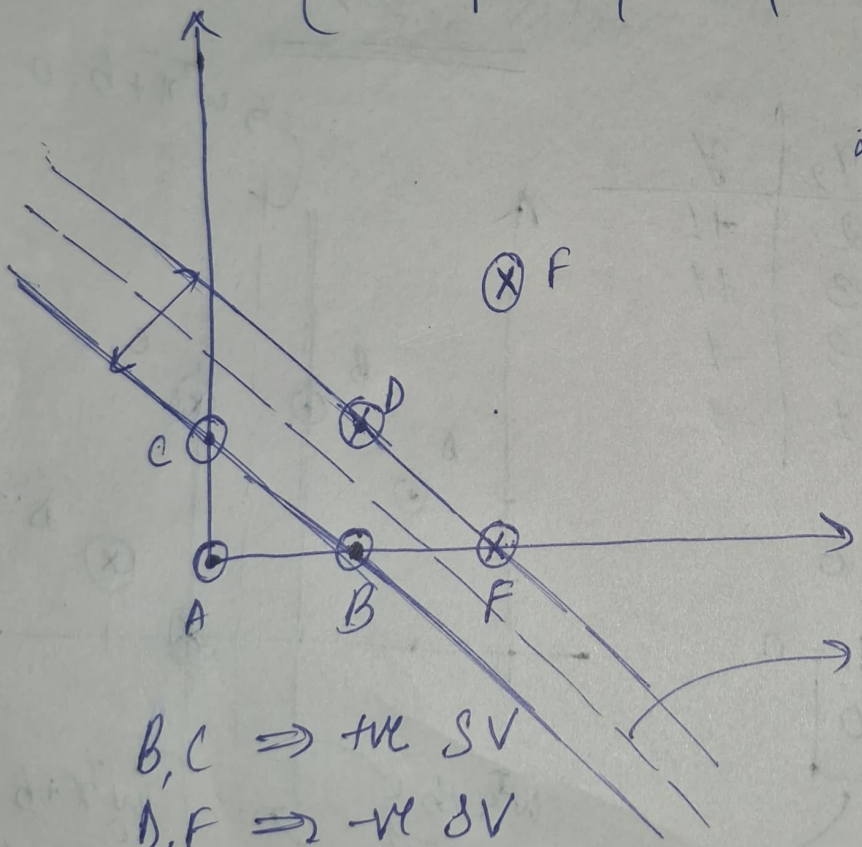
$0.2 \longrightarrow 0.255$

$b_2 =$

$0.6 \longrightarrow 0.655$

(b)

	Class	x_1	x_2	Label	
⊙	+	0	0	+	A
⊙	+	1	0	+	B
⊙	+	0	1	+	C
⊗	−	1	1	−	D
⊗	−	2	2	−	E
⊗	−	2	0	−	F



linearly separable classes.

$w^T x + b = 1$ {+ve SV}

$w^T x + b = -1$ {−ve SV}

$w^T x + b = 0$ {hyperplane}

more margin hyperplane

B, C ⇒ +ve SV

D, F ⇒ −ve SV

Solving for most optimal hyperplane using B, C, D, F

B: $w_1 + 0.w_2 + b = 1 \Rightarrow w_1 + b = 1$

C: $w_2 + b = 1$

D: $w_1 + w_2 + b = -1$

F: $2w_1 + b = -1$

$$\begin{cases} w_1 + b = 1 \\ 2w_1 + b = -1 \end{cases}$$

$w_1 = -2$

$w_2 + b = 1$

$b = 3$

$w_1 + w_2 + b = -1$

$w_2 = -2$

Hyperplane:

$-2x_1 - 2x_2 + 3 = 0$

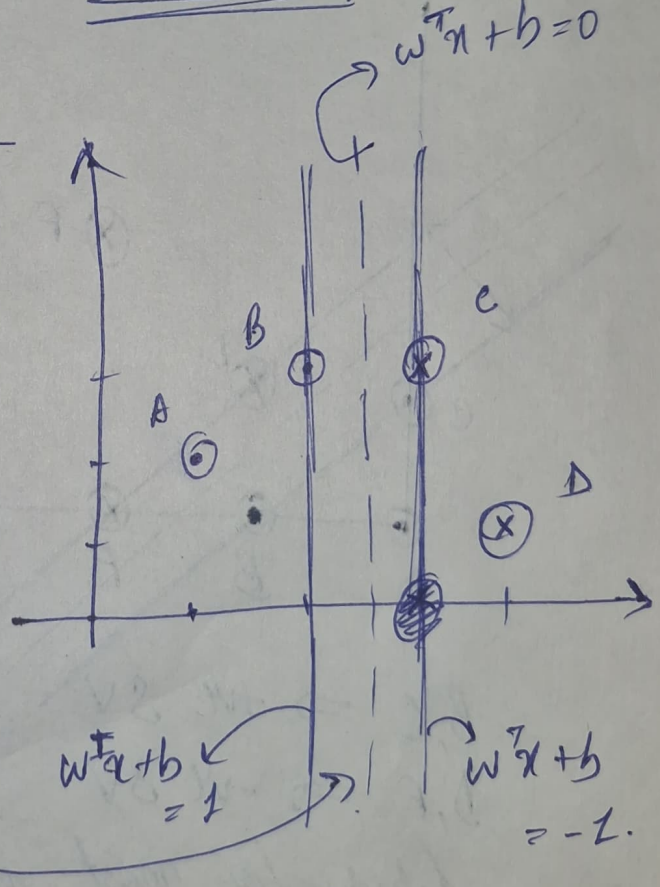
SV vectors:

+ : B, C

− : D, F

(c)

	x_1	x_2	y
A	1	2	+1
B	2	3	+1
C	3	3	−1
D	4	1	−1



$wx + b = 0$

$w_1 x_1 + w_2 x_2 + b = 0$

$x_1 - x_2 = 0$

$z = -2x_1 + 5 = 0$

(i) Margin (assuming margin is the distance from the +ve SV to the −ve)

$$\text{Margin} = \frac{2}{\|w\|} = \frac{2}{\sqrt{-2^2 + 0^2}}$$

$$= \frac{1}{}$$

(ii) Points with $w^T x + b = \pm 1$ are the SV

$-2(1)+5 = 3$

$-2(2)+5 = 1$ } SV → (2,3) : +ve

$-2(3)+5 = -1$ } SV → (3,3) : -ve

$-2(4)+5 = -3$

(iii) $x_1 = 1, x_2 = 3$

$y_{pred}(w^T x + b) \geq 1$ $\quad y_{pred} \in \{-1, 1\}$

$\therefore y_{pred}(-2 \times 1 + 0 \times 3 + 5) \geq 1$

$y_{pred}(3) \geq 1$

$y_{pred} \geq 1/3$ $\Rightarrow y_{pred} = +1$

$\Rightarrow$ +ve class

In [3]:
```
import numpy as np
from sklearn.model_selection import train_test_split
import struct
import matplotlib.pyplot as plt
import random
import pickle
```

Neural Network

In [ ]:
```
class NeuralNet:
    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_prime(self, x):
        return self.sigmoid(x) * (1 - self.sigmoid(x))

    def relu(self, x):
        return np.maximum(0, x)

    def relu_prime(self, x):
        return np.where(x > 0, 1, 0)

    def tanh(self, x):
        return np.tanh(x)

    def tanh_prime(self, x):
        return 1 - np.tanh(x)**2

    def leaky_relu(self, x):
        return np.where(x > 0, x, x * 0.01)

    def leaky_relu_prime(self, x):
        return np.where(x > 0, 1, 0.01)

    def softmax(self, x):
        exps = np.exp(x - np.max(x))
        exps = np.where(exps == 0, 1e-10, exps)
        return exps / np.sum(exps, axis=1, keepdims=True)

    def softmax_prime(self, x):
        return self.softmax(x) * (1 - self.softmax(x))

    def random_weight(self, x, y):
        if self.activation == self.relu or self.activation == self.leaky_relu:
            limit = np.sqrt(6 / x)
            return np.random.uniform(-limit, limit, (x, y)).astype(np.float64)
        return np.random.uniform(-1, 1, (x, y)).astype(np.float64)

    def normal_weight(self, x, y):
        if self.activation == self.relu or self.activation == self.leaky_relu:
            return np.random.randn(x, y) * np.sqrt(2 / x)
        return np.random.randn(x, y).astype(np.float64)

    def zero_weight(self, x, y):
        return np.zeros((x, y), dtype=np.float64)
```

```
    def __init__(self, n_hid, hiddenlayers, input_len, output_len, activation, w
        self.n_hid = n_hid
        self.hiddenlayers = hiddenlayers
        self.input_len = input_len
        self.output_len = output_len
        self.learn_rate = learn_rate
        self.epochs = epochs
        self.batch_size = batch_size
        self.early_stopping = early_stopping

        if weight_init == 'random':
            self.weight_init = self.random_weight
        elif weight_init == 'normal':
            self.weight_init = self.normal_weight
        elif weight_init == 'zero':
            self.weight_init = self.zero_weight
        else:
            raise ValueError('Invalid weight initialization')

        if activation == 'relu':
            self.activation = self.relu
            self.activation_prime = self.relu_prime
        elif activation == 'sigmoid':
            self.activation = self.sigmoid
            self.activation_prime = self.sigmoid_prime
        elif activation == 'tanh':
            self.activation = self.tanh
            self.activation_prime = self.tanh_prime
        elif activation == 'leaky_relu':
            self.activation = self.leaky_relu
            self.activation_prime = self.leaky_relu_prime
        else:
            raise ValueError('Invalid activation function')

        self.weights = []
        self.biases = []

        self.weights.append(self.weight_init(self.input_len, self.hiddenlayers[0
        self.biases.append(self.weight_init(1, self.hiddenlayers[0]))

        for i in range(1, len(self.hiddenlayers)):
            self.weights.append(self.weight_init(self.hiddenlayers[i - 1], self.
            self.biases.append(self.weight_init(1, self.hiddenlayers[i]))

        self.weights.append(self.weight_init(self.hiddenlayers[-1], self.output_
        self.biases.append(self.weight_init(1, self.output_len))

    def feedforward(self, x):                           # forward pass, returns
        act = x
        self.acts = [x]
        self.zs = []                            # z = w * x + b

        for i in range(len(self.weights) - 1):
            z = np.dot(act, self.weights[i]) + self.biases[i]
            act = self.activation(z)
            self.acts.append(act)
            self.zs.append(z)

        z = np.dot(act, self.weights[-1]) + self.biases[-1]         # softmax
        act = self.softmax(z)
```

```
        self.acts.append(act)
        self.zs.append(z)

        return act

    def loss(self, pred_proba, y):                      # cross entropy loss
        ce = np.zeros(pred_proba.shape[0])
        for i in range(len(y)):
            ce[i] = -np.log(pred_proba[i][y[i]])
        sum = np.sum(ce)

        return sum / ce.shape[0]

    def backpropagation(self, prob, y):
        delta = (prob - np.eye(self.output_len)[y])
        self.weights[-1] -= self.learn_rate * np.dot(self.acts[-2].T, delta) / s
        self.biases[-1] -= self.learn_rate * np.mean(delta, axis=0)

        for i in range(len(self.weights) - 2, -1, -1):
            delta = np.dot(delta, self.weights[i+1].T) * self.activation_prime(s
            self.weights[i] -= self.learn_rate * np.dot(self.acts[i].T, delta) /
            self.biases[i] -= self.learn_rate * np.mean(delta, axis=0)

    def fit(self, x, y, x_val, y_val):
        self.train_loss = []
        self.val_loss = []
        for epoch in range(1,self.epochs + 1):
            for i in range(0, len(x), self.batch_size):
                x_batch = x[i:i+self.batch_size]
                y_batch = y[i:i+self.batch_size]

                prob = self.feedforward(x_batch)
                self.backpropagation(prob, y_batch)

            self.train_loss.append(self.loss(self.feedforward(x), y))
            self.val_loss.append(self.loss(self.feedforward(x_val), y_val))

            if epoch % 10 == 0:
                print('epoch:', epoch, 'train loss:', self.train_loss[-1], 'val
            # early stopping
            if self.early_stopping and epoch > 50 and np.mean(self.train_loss[-6
                print('early stopping at epoch:', epoch)
                break

    def predict(self, x):
        return np.argmax(self.feedforward(x), axis=1)

    def predict_proba(self, x):
        return self.feedforward(x)

    def score(self, x, y):
        return np.mean(self.predict(x) == y)
```

Loading data

In [16]:
```
def load_idx(file_path):
    """Load an MNIST `.idx` format file."""
    with open(file_path, 'rb') as f:
```

```
        magic, num_items = struct.unpack(">II", f.read(8))
        if magic == 2051:  # Images
            rows, cols = struct.unpack(">II", f.read(8))
            data = np.fromfile(f, dtype=np.uint8).reshape(num_items, rows * cols
        elif magic == 2049:  # Labels
            data = np.fromfile(f, dtype=np.uint8)
    return data
```

In [20]:
```
train_images = load_idx('data/train-images.idx3-ubyte')
train_labels = load_idx('data/train-labels.idx1-ubyte')
test_images = load_idx('data/t10k-images.idx3-ubyte')
test_labels = load_idx('data/t10k-labels.idx1-ubyte')
```

In [ ]:
```
np.random.seed(42)

x = np.concatenate([train_images, test_images])
y = np.concatenate([train_labels, test_labels])

idx = np.random.permutation(len(x))
x = x[idx]
y = y[idx]

n = len(x)
n_train = int(n * 0.8)
n_val = int(n * 0.1)
n_test = n - n_train - n_val

# train val test split
x_train = x[:n_train]
y_train = y[:n_train]
x_val = x[n_train:n_train+n_val]
y_val = y[n_train:n_train+n_val]
x_test = x[n_train+n_val:]
y_test = y[n_train+n_val:]
```

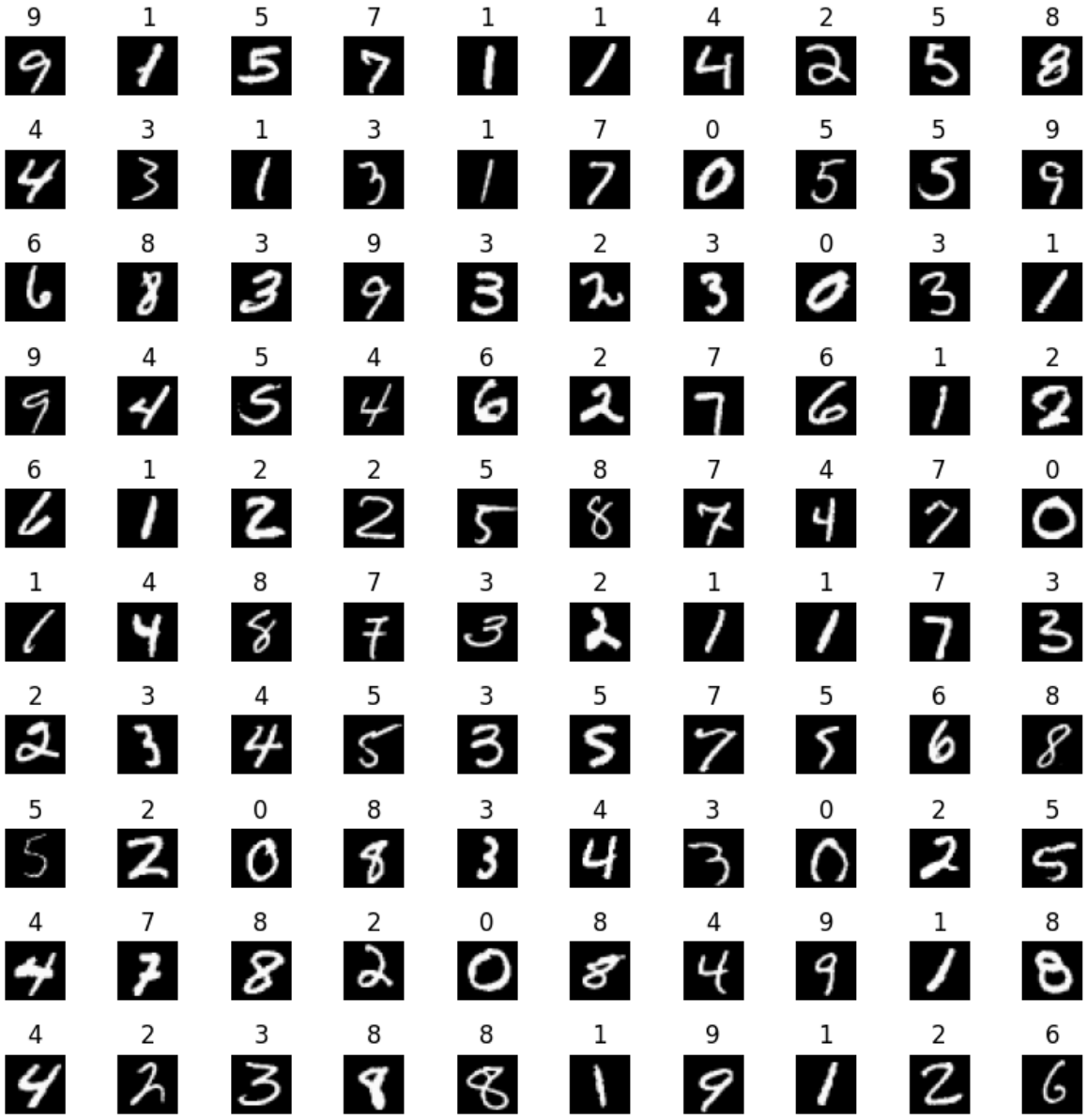
In [22]:
```
x_train = x_train.reshape(-1, 28*28)
x_test = x_test.reshape(-1, 28*28)
x_val = x_val.reshape(-1, 28*28)
x_train.shape, y_train.shape, x_test.shape, y_test.shape, x_val.shape, y_val.sha
```

Out[22]:
```
((56000, 784), (56000,), (7000, 784), (7000,), (7000, 784), (7000,))
```

In [ ]:
```
# normalize to be between 0 and 1
x_train = x_train / 255.0
x_test = x_test / 255.0
x_val = x_val / 255.0
```

In [ ]:
```
# plotting the data
def plot_images(images, labels, rows=10, cols=10):
    fig, axes = plt.subplots(rows, cols, figsize=(8, 8))
    for i in range(rows):
        for j in range(cols):
            index = random.randint(0, len(images))
            axes[i, j].imshow(images[index].reshape(28, 28), cmap='gray')
            axes[i, j].set_title(labels[index])
            axes[i, j].axis('off')
    plt.tight_layout()
    plt.show()
```

```
plot_images(x_train, y_train)
```



Training the data

Relu + Random

In [58]:
```
relu_random = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=784,
relu_random.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.302533279397232 val loss: 2.3011685119274063
epoch: 20 train loss: 2.3024096668184635 val loss: 2.3010382362396062
epoch: 30 train loss: 2.30237971939461 val loss: 2.300976636243327
epoch: 40 train loss: 2.3023161062468525 val loss: 2.3009154997942547
epoch: 50 train loss: 2.3023998809029753 val loss: 2.3009345787992928
epoch: 60 train loss: 2.3024260662286475 val loss: 2.300946446708343
epoch: 70 train loss: 2.302504311224289 val loss: 2.3010070834683005
epoch: 80 train loss: 2.3025946648577795 val loss: 2.301117263179714
epoch: 90 train loss: 2.302925065263403 val loss: 2.3014146410208616
epoch: 100 train loss: 2.3035621181206247 val loss: 2.3020266374049525
```

In [59]:
```
print('Train accuracy:', relu_random.score(x_train, y_train))
print('Validation accuracy:', relu_random.score(x_val, y_val))
print('Test accuracy:', relu_random.score(x_test, y_test))
```

```
Train accuracy: 0.10226785714285715
Validation accuracy: 0.10671428571428572
Test accuracy: 0.09528571428571428
```

In [84]:
```
# save model to pickle
with open('relu_random.pkl', 'wb') as f:
    pickle.dump(relu_random, f)
```

Relu + Normal

In [60]:
```
relu_normal = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=784,
relu_normal.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.3031923736987294 val loss: 2.302188776570358
epoch: 20 train loss: 2.3031209860169333 val loss: 2.3021176459379644
epoch: 30 train loss: 2.303079478034578 val loss: 2.3020656276509768
epoch: 40 train loss: 2.303047402860048 val loss: 2.3020211283262437
epoch: 50 train loss: 2.303025590657762 val loss: 2.3019846173442198
epoch: 60 train loss: 2.303014935501075 val loss: 2.3019571084144688
epoch: 70 train loss: 2.303017118856237 val loss: 2.3019400791687668
epoch: 80 train loss: 2.3030341031824806 val loss: 2.3019356388818246
epoch: 90 train loss: 2.3031010418471127 val loss: 2.301981930172968
epoch: 100 train loss: 2.3031565775632035 val loss: 2.302012455070436
```

In [61]:
```
print('Train accuracy:', relu_normal.score(x_train, y_train))
print('Validation accuracy:', relu_normal.score(x_val, y_val))
print('Test accuracy:', relu_normal.score(x_test, y_test))
```

```
Train accuracy: 0.11185714285714286
Validation accuracy: 0.11228571428571428
Test accuracy: 0.11814285714285715
```

In [85]:
```
with open('relu_normal.pkl', 'wb') as f:
    pickle.dump(relu_normal, f)
```

Relu + Zero

In [62]:
```
relu_zero = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=784, c
relu_zero.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.3014200255569803 val loss: 2.3011710772553635
epoch: 20 train loss: 2.3012219643465484 val loss: 2.3008656359439903
epoch: 30 train loss: 2.3011882473158125 val loss: 2.3007847135428743
epoch: 40 train loss: 2.3011824032742796 val loss: 2.3007579402177765
epoch: 50 train loss: 2.301181364030809 val loss: 2.3007475851006634
epoch: 60 train loss: 2.301181173544364 val loss: 2.3007432401383796
epoch: 70 train loss: 2.301181137416241 val loss: 2.3007413494620907
epoch: 80 train loss: 2.301181130317566 val loss: 2.300740514015715
epoch: 90 train loss: 2.3011811288886235 val loss: 2.3007401424513314
epoch: 100 train loss: 2.30118112860864 val loss: 2.3007399767290355
```

In [63]:
```
print('Train accuracy:', relu_zero.score(x_train, y_train))
print('Validation accuracy:', relu_zero.score(x_val, y_val))
print('Test accuracy:', relu_zero.score(x_test, y_test))
```

```
Train accuracy: 0.11185714285714286
Validation accuracy: 0.11228571428571428
Test accuracy: 0.11814285714285715
```

In [86]:
```
with open('relu_zero.pkl', 'wb') as f:
    pickle.dump(relu_zero, f)
```

Sigmoid + Random

In [65]:
```
sigmoid_random = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=7
sigmoid_random.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.3000895871059135 val loss: 2.2995649370940305
epoch: 20 train loss: 2.2997609040073166 val loss: 2.2992364568872055
epoch: 30 train loss: 2.2994112274886622 val loss: 2.298886646374365
epoch: 40 train loss: 2.2990372589709813 val loss: 2.2985124450109895
epoch: 50 train loss: 2.2986362430341707 val loss: 2.298111338593532
epoch: 60 train loss: 2.2982071489311715 val loss: 2.2976824938279656
epoch: 70 train loss: 2.2977519543018117 val loss: 2.2972279476384507
epoch: 80 train loss: 2.2972759910128095 val loss: 2.2967528760675733
epoch: 90 train loss: 2.2967863130898247 val loss: 2.296263968595581
epoch: 100 train loss: 2.296288727474775 val loss: 2.2957665558201197
```

In [66]:
```
print('Train accuracy:', sigmoid_random.score(x_train, y_train))
print('Validation accuracy:', sigmoid_random.score(x_val, y_val))
print('Test accuracy:', sigmoid_random.score(x_test, y_test))
```

```
Train accuracy: 0.11217857142857143
Validation accuracy: 0.11257142857142857
Test accuracy: 0.11828571428571429
```

In [87]:
```
with open('sigmoid_random.pkl', 'wb') as f:
    pickle.dump(sigmoid_random, f)
```

Sigmoid + Normal

In [67]:
```
sigmoid_normal = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=7
sigmoid_normal.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.280429186367357 val loss: 2.2787577534286365
epoch: 20 train loss: 2.2648998416906423 val loss: 2.2629124996632606
epoch: 30 train loss: 2.249850077951818 val loss: 2.2475957288862776
epoch: 40 train loss: 2.2361984707378975 val loss: 2.233857379676308
epoch: 50 train loss: 2.2217235541760663 val loss: 2.2192894894257162
epoch: 60 train loss: 2.2041926473091893 val loss: 2.2015768690379622
epoch: 70 train loss: 2.1837622028240147 val loss: 2.1809837047509437
epoch: 80 train loss: 2.1619285519546616 val loss: 2.1590308827584535
epoch: 90 train loss: 2.1375625995362286 val loss: 2.13458560498864
epoch: 100 train loss: 2.1086832916067366 val loss: 2.1058014444388866
```

In [68]:
```
print('Train accuracy:', sigmoid_normal.score(x_train, y_train))
print('Validation accuracy:', sigmoid_normal.score(x_val, y_val))
print('Test accuracy:', sigmoid_normal.score(x_test, y_test))
```

```
Train accuracy: 0.3604107142857143
Validation accuracy: 0.363
Test accuracy: 0.357
```

In [88]:
```
with open('sigmoid_normal.pkl', 'wb') as f:
    pickle.dump(sigmoid_normal, f)
```

Sigmoid + Zero

In [69]:
```
sigmoid_zero = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=784
sigmoid_zero.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.3011897389794025 val loss: 2.300726675921646
epoch: 20 train loss: 2.301189742459215 val loss: 2.300726366388017
epoch: 30 train loss: 2.3011897445602902 val loss: 2.300726364322174
epoch: 40 train loss: 2.30118974665454 val loss: 2.3007263624685286
epoch: 50 train loss: 2.3011897487496893 val loss: 2.300726360613887
epoch: 60 train loss: 2.301189750845744 val loss: 2.300726358758094
epoch: 70 train loss: 2.3011897529427063 val loss: 2.300726356901152
epoch: 80 train loss: 2.301189755040576 val loss: 2.3007263550430594
epoch: 90 train loss: 2.301189757139352 val loss: 2.300726353183817
epoch: 100 train loss: 2.3011897592390356 val loss: 2.3007263513234264
```

In [70]:
```
print('Train accuracy:', sigmoid_zero.score(x_train, y_train))
print('Validation accuracy:', sigmoid_zero.score(x_val, y_val))
print('Test accuracy:', sigmoid_zero.score(x_test, y_test))
```

```
Train accuracy: 0.11185714285714286
Validation accuracy: 0.11228571428571428
Test accuracy: 0.11814285714285715
```

In [89]:
```
with open('sigmoid_zero.pkl', 'wb') as f:
    pickle.dump(sigmoid_zero, f)
```

Tanh + Random

In [71]:
```
tanh_random = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=784,
tanh_random.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 1.4473753615596472 val loss: 1.445099489926801
epoch: 20 train loss: 1.0885904549994097 val loss: 1.088733078789408
epoch: 30 train loss: 0.9251660518790438 val loss: 0.9272702869896307
epoch: 40 train loss: 0.8309379212518876 val loss: 0.8341754463014588
epoch: 50 train loss: 0.7523618386809624 val loss: 0.7545164049190591
epoch: 60 train loss: 0.6820148632163359 val loss: 0.682665538523918
epoch: 70 train loss: 0.6236371645120913 val loss: 0.6229122882755281
epoch: 80 train loss: 0.5796278304617615 val loss: 0.5784631289503872
epoch: 90 train loss: 0.5449952582218989 val loss: 0.5442908960454294
epoch: 100 train loss: 0.5145095561119645 val loss: 0.5148261687463348
```

In [72]:
```
print('Train accuracy:', tanh_random.score(x_train, y_train))
print('Validation accuracy:', tanh_random.score(x_val, y_val))
print('Test accuracy:', tanh_random.score(x_test, y_test))
```

```
Train accuracy: 0.8492678571428571
Validation accuracy: 0.8498571428571429
Test accuracy: 0.8488571428571429
```

In [90]:
```
with open('tanh_random.pkl', 'wb') as f:
    pickle.dump(tanh_random, f)
```

Tanh + Normal

In [73]:
```
tanh_normal = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=784,
tanh_normal.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 1.577467337850735 val loss: 1.592685595444883
epoch: 20 train loss: 1.3907133073939428 val loss: 1.4041128059496883
epoch: 30 train loss: 1.2124523035098562 val loss: 1.223060840769209
epoch: 40 train loss: 1.1181969552049502 val loss: 1.1280977567055719
epoch: 50 train loss: 1.036416043338113 val loss: 1.0452304457876702
epoch: 60 train loss: 0.9598918751688001 val loss: 0.9703528991330452
epoch: 70 train loss: 0.9123725450602539 val loss: 0.925418591174799
epoch: 80 train loss: 0.8770554737331994 val loss: 0.8921355889604874
epoch: 90 train loss: 0.8380388954810623 val loss: 0.8540691670995921
epoch: 100 train loss: 0.8006619997094087 val loss: 0.8168014995875938
```

In [74]:
```
print('Train accuracy:', tanh_normal.score(x_train, y_train))
print('Validation accuracy:', tanh_normal.score(x_val, y_val))
print('Test accuracy:', tanh_normal.score(x_test, y_test))
```

```
Train accuracy: 0.7513392857142858
Validation accuracy: 0.7471428571428571
Test accuracy: 0.7485714285714286
```

In [91]:
```
with open('tanh_normal.pkl', 'wb') as f:
    pickle.dump(tanh_normal, f)
```

Tanh + Zero

In [75]:
```
tanh_zero = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=784, c
tanh_zero.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.3014200255569803 val loss: 2.3011710772553635
epoch: 20 train loss: 2.3012219643465484 val loss: 2.3008656359439903
epoch: 30 train loss: 2.3011882473158125 val loss: 2.3007847135428743
epoch: 40 train loss: 2.3011824032742796 val loss: 2.3007579402177765
epoch: 50 train loss: 2.301181364030809 val loss: 2.3007475851006634
epoch: 60 train loss: 2.301181173544364 val loss: 2.3007432401383796
epoch: 70 train loss: 2.301181137416241 val loss: 2.3007413494620907
epoch: 80 train loss: 2.301181130317566 val loss: 2.300740514015715
epoch: 90 train loss: 2.3011811288886235 val loss: 2.3007401424513314
epoch: 100 train loss: 2.30118112860864 val loss: 2.3007399767290355
```

In [76]:
```
print('Train accuracy:', tanh_zero.score(x_train, y_train))
print('Validation accuracy:', tanh_zero.score(x_val, y_val))
print('Test accuracy:', tanh_zero.score(x_test, y_test))
```

```
Train accuracy: 0.11185714285714286
Validation accuracy: 0.11228571428571428
Test accuracy: 0.11814285714285715
```

In [92]:
```
with open('tanh_zero.pkl', 'wb') as f:
    pickle.dump(tanh_zero, f)
```

Leaky Relu + Random

In [77]:
```
leaky_relu_random = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_le
leaky_relu_random.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.305032891809814 val loss: 2.304054459616786
epoch: 20 train loss: 2.3050442162887195 val loss: 2.304055441300012
epoch: 30 train loss: 2.3050870982267813 val loss: 2.3040862456692284
epoch: 40 train loss: 2.305163419976106 val loss: 2.304146985027259
epoch: 50 train loss: 2.305249572459818 val loss: 2.3042135057030486
epoch: 60 train loss: 2.305401299072673 val loss: 2.304340028012526
epoch: 70 train loss: 2.3055938500580875 val loss: 2.304499304279268
epoch: 80 train loss: 2.3059214734106677 val loss: 2.304820214102725
epoch: 90 train loss: 2.3060799263432883 val loss: 2.3049015369221166
epoch: 100 train loss: 2.3064091342350768 val loss: 2.3051522981079895
```

In [78]:
```
print('Train accuracy:', leaky_relu_random.score(x_train, y_train))
print('Validation accuracy:', leaky_relu_random.score(x_val, y_val))
print('Test accuracy:', leaky_relu_random.score(x_test, y_test))
```

```
Train accuracy: 0.10226785714285715
Validation accuracy: 0.10671428571428572
Test accuracy: 0.09528571428571428
```

In [93]:
```
with open('leaky_relu_random.pkl', 'wb') as f:
    pickle.dump(leaky_relu_random, f)
```

Leaky Relu + Normal

In [79]:
```
leaky_relu_normal = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_le
leaky_relu_normal.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.3043021284523832 val loss: 2.3036060047096454
epoch: 20 train loss: 2.3042507862436286 val loss: 2.3035367896872283
epoch: 30 train loss: 2.3042064973174035 val loss: 2.303474299360565
epoch: 40 train loss: 2.304172885011064 val loss: 2.3034211110987926
epoch: 50 train loss: 2.304156654427881 val loss: 2.303379070236677
epoch: 60 train loss: 2.304139753066971 val loss: 2.303344539929709
epoch: 70 train loss: 2.3041408336648255 val loss: 2.3033220886017576
epoch: 80 train loss: 2.3041547885332934 val loss: 2.303311257676663
epoch: 90 train loss: 2.3042139854867925 val loss: 2.3033523524666313
epoch: 100 train loss: 2.3042519404248036 val loss: 2.3033633302814938
```

In [80]:
```
print('Train accuracy:', leaky_relu_normal.score(x_train, y_train))
print('Validation accuracy:', leaky_relu_normal.score(x_val, y_val))
print('Test accuracy:', leaky_relu_normal.score(x_test, y_test))
```

```
Train accuracy: 0.10226785714285715
Validation accuracy: 0.10671428571428572
Test accuracy: 0.09528571428571428
```

In [94]:
```
with open('leaky_relu_normal.pkl', 'wb') as f:
    pickle.dump(leaky_relu_normal, f)
```

Leaky Relu + Zero

In [99]:
```
leaky_relu_zero = NeuralNet(n_hid=4, hiddenlayers=[256, 128, 64, 32], input_len=
leaky_relu_zero.fit(x_train, y_train, x_val, y_val)
```

```
epoch: 10 train loss: 2.3037254956891218 val loss: 2.302475096546917
epoch: 20 train loss: 2.303767426890588 val loss: 2.3024912049730433
epoch: 30 train loss: 2.303880533387858 val loss: 2.3025558662601937
epoch: 40 train loss: 2.3040095208505087 val loss: 2.3026993442307515
epoch: 50 train loss: 2.3042047107792034 val loss: 2.3028435208804305
epoch: 60 train loss: 2.3044750957577427 val loss: 2.303054455380769
epoch: 70 train loss: 2.304810701940396 val loss: 2.3033471265959267
epoch: 80 train loss: 2.305313607592774 val loss: 2.3037729508067772
epoch: 90 train loss: 2.3060154405123767 val loss: 2.3043836849119965
epoch: 100 train loss: 2.307023701308522 val loss: 2.305282351320243
```

In [100…

```
print('Train accuracy:', leaky_relu_zero.score(x_train, y_train))
print('Validation accuracy:', leaky_relu_zero.score(x_val, y_val))
print('Test accuracy:', leaky_relu_zero.score(x_test, y_test))
```

```
Train accuracy: 0.10226785714285715
Validation accuracy: 0.10671428571428572
Test accuracy: 0.09528571428571428
```

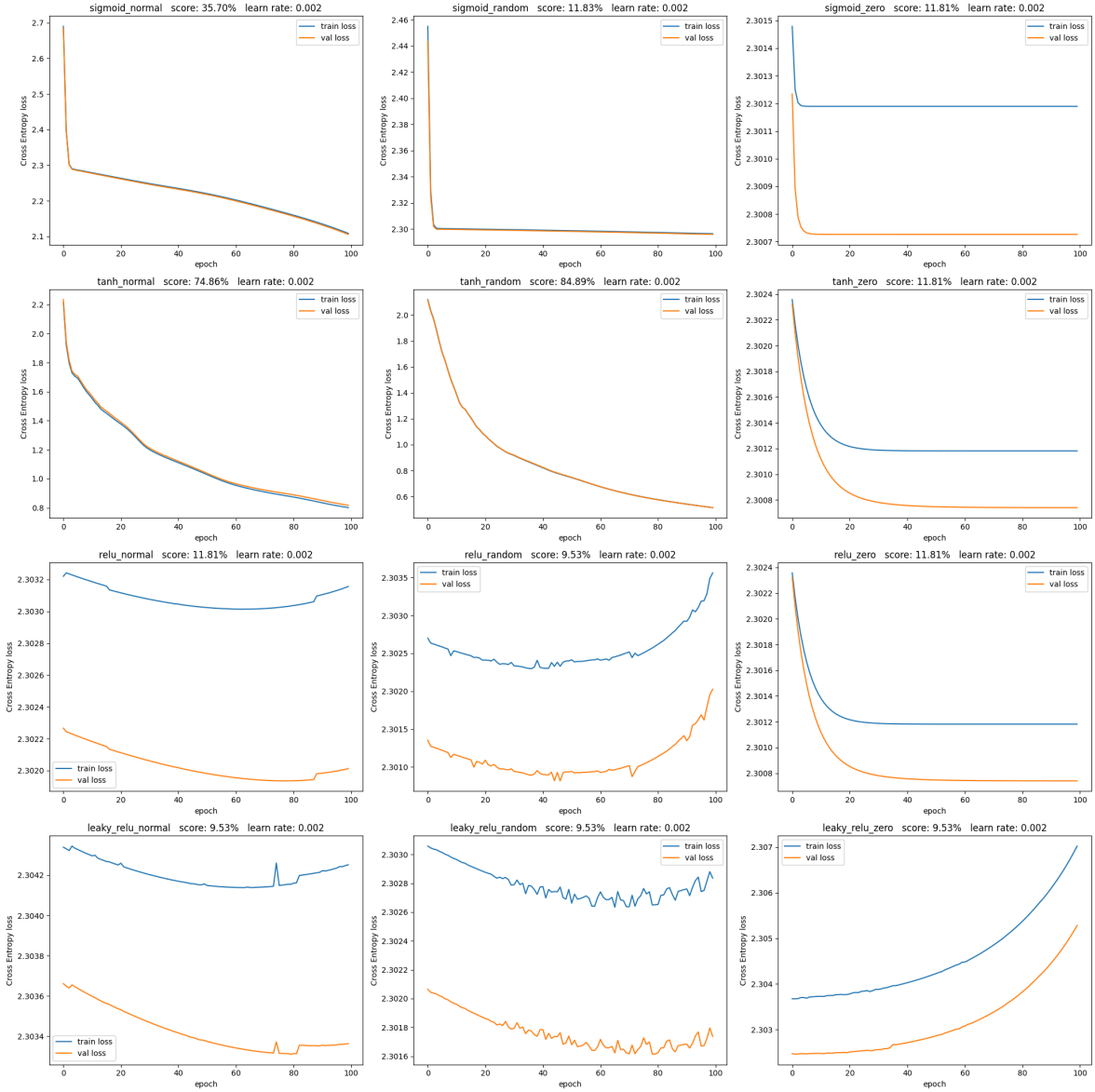
In [101…

```
with open('leaky_relu_zero.pkl', 'wb') as f:
    pickle.dump(leaky_relu_zero, f)
```

Plotting the loss curves

In [ ]:

```
activation = ['sigmoid', 'tanh', 'relu', 'leaky_relu']
weight_init = ['normal', 'random', 'zero']


plt.figure(figsize=(20, 20))
for act in activation:
    for weight in weight_init:
        model = pickle.load(open(f'{act}_{weight}.pkl', 'rb'))

        plt.subplot(4, 3, activation.index(act) * 3 + weight_init.index(weight)
        plt.plot(model.train_loss, label='train loss')
        plt.plot(model.val_loss, label='val loss')
        plt.title(f'{act}_{weight}   ' + f'score: {model.score(x_test, y_test) *
        plt.xlabel('epoch')
        plt.ylabel('Cross Entropy loss')
        plt.legend()

        del model

plt.tight_layout()
plt.show()
```



- All models were run at most 100 epochs and batch size 128.
- Learning rate = 0.002 and 'Adam' solver
- ReLU and Leaky ReLU activation functions are highly sensitive to initialization of weights and learning rate.
- From the plots tanh seemed to perform the best given same initialisation and learning rates for all.

In [ ]:

In [13]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss
from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import accuracy_score
from sklearn.utils.extmath import safe_sparse_dot
```

Data Loading

In [2]:
```
train_data = pd.read_csv("Fashion-MNIST/fashion-mnist_train.csv").head(8000)
test_data = pd.read_csv("Fashion-MNIST/fashion-mnist_test.csv").head(2000)
```

In [3]:
```
train_data.head()
```

Out[3]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel7
0	2	0	0	0	0	0	0	0	0	0	...	
1	9	0	0	0	0	0	0	0	0	0	...	
2	6	0	0	0	0	0	0	0	5	0	...	
3	0	0	0	0	1	2	0	0	0	0	...	
4	3	0	0	0	0	0	0	0	0	0	...	

5 rows × 785 columns

In [5]:
```
test_data.head()
```

Out[5]:

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel7
0	0	0	0	0	0	0	0	0	9	8	...	
1	1	0	0	0	0	0	0	0	0	0	...	
2	2	0	0	0	0	0	0	14	53	99	...	
3	2	0	0	0	0	0	0	0	0	0	...	
4	3	0	0	0	0	0	0	0	0	0	...	

5 rows × 785 columns

Preprocessing

In [ ]:
```
# Normalisation

X_train = train_data.drop(columns=['label']).values / 255.0
y_train = train_data['label'].values
```

```
X_test = test_data.drop(columns=['label']).values / 255.0
y_test = test_data['label'].values
```

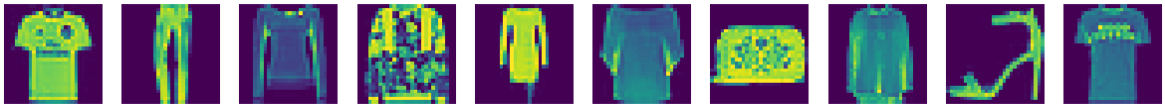
In [17]:
```
X_train
```

Out[17]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=int64)
```

In [18]:
```
y_train
```

Out[18]:
```
array([2, 9, 6, ..., 6, 4, 8], dtype=int64)
```

In [ ]:
```
fig, axes = plt.subplots(1, 10, figsize=(15, 5))
for i, ax in enumerate(axes):
    ax.imshow(X_test[i].reshape(28, 28))
    ax.axis('off')
plt.show()
```



MLP Classifier

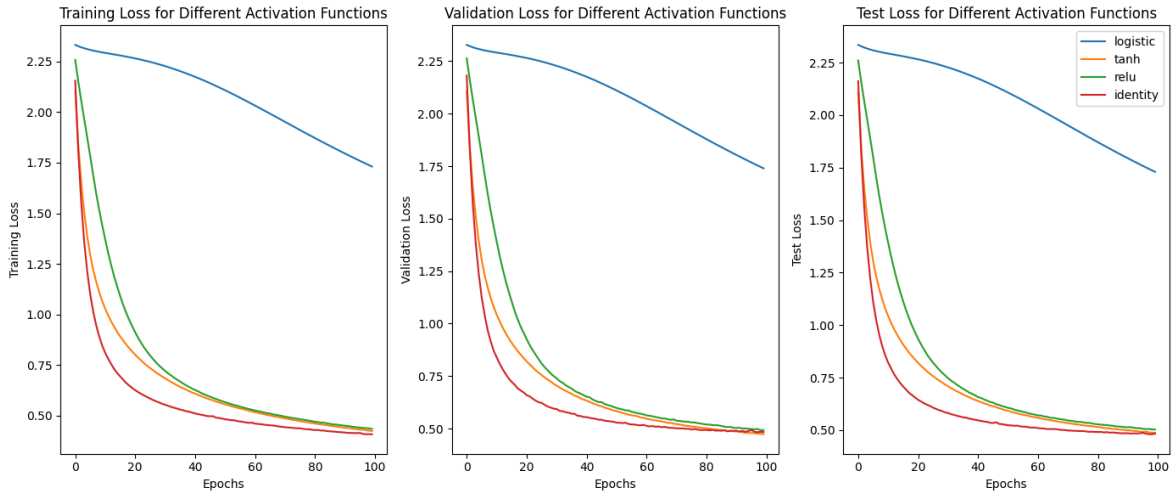
In [5]:
```
X_train_split, X_val_split, y_train_split, y_val_split = train_test_split(X_trai
```

In [ ]:
```
# activation functions and dictionary to store loss histories
activations = ['logistic', 'tanh', 'relu', 'identity']
loss_history = {}
accuracies = {}

# Loop over each activation function
for activation in activations:
    mlp = MLPClassifier(hidden_layer_sizes=(128, 64, 32), activation=activation,
                        solver='adam', max_iter=1, batch_size=128,
                        learning_rate_init=2e-5, warm_start=True)

    # Track loss for each epoch
    training_loss = []
    validation_loss = []
    test_loss = []

    training_accuracy = []
    validation_accuracy = []
    test_accuracy = []

    # Train model for 100 epochs manually to log losses
    for epoch in range(100):
        mlp.fit(X_train_split, y_train_split)  # Train for one epoch

        # Calculate training loss
        y_train_pred = mlp.predict_proba(X_train_split)
```

```
        train_loss = log_loss(y_train_split, y_train_pred)
        training_loss.append(train_loss)

        # Calculate training accuracy
        y_train_pred = mlp.predict(X_train_split)
        train_accuracy = accuracy_score(y_train_split, y_train_pred)
        training_accuracy.append(train_accuracy)

        # Calculate validation loss
        y_val_pred = mlp.predict_proba(X_val_split)
        val_loss = log_loss(y_val_split, y_val_pred)
        validation_loss.append(val_loss)

        # Calculate validation accuracy
        y_val_pred = mlp.predict(X_val_split)
        val_accuracy = accuracy_score(y_val_split, y_val_pred)
        validation_accuracy.append(val_accuracy)

        # Calculate test loss
        y_test_pred = mlp.predict_proba(X_test)
        test_loss.append(log_loss(y_test, y_test_pred))

        # Calculate test accuracy
        y_test_pred = mlp.predict(X_test)
        test_accuracy.append(accuracy_score(y_test, y_test_pred))

    # Store loss history for this activation function
    loss_history[activation] = {'train_loss': training_loss, 'val_loss': validat
    accuracies[activation] = {'train_accuracy': training_accuracy, 'val_accuracy
```

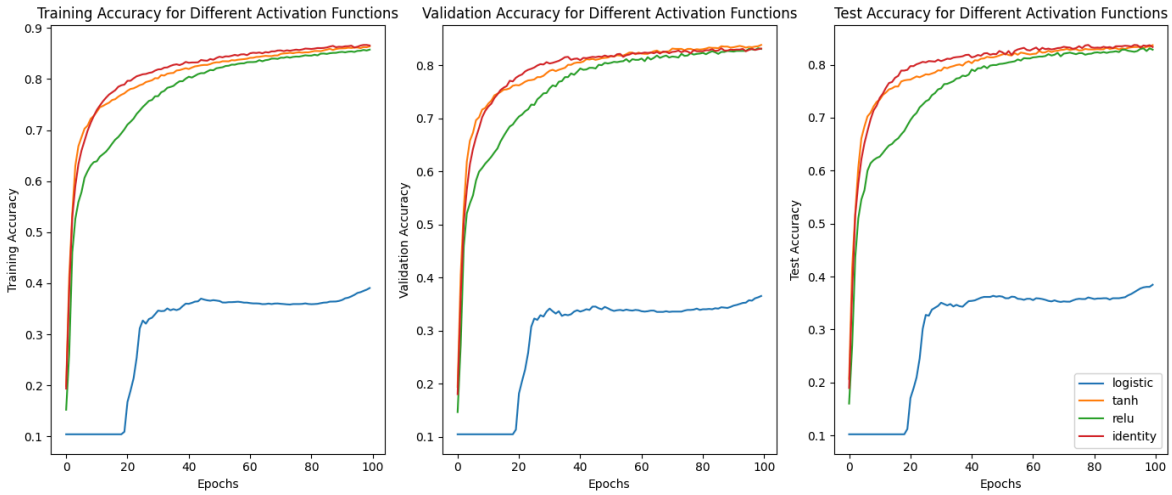
In [18]:

```
plt.figure(figsize=(14, 6))
plt.subplot(1, 3, 1)
for activation in activations:
    plt.plot(loss_history[activation]['train_loss'], label=activation)
plt.xlabel('Epochs')
plt.ylabel('Training Loss')
plt.title('Training Loss for Different Activation Functions')

plt.subplot(1, 3, 2)
for activation in activations:
    plt.plot(loss_history[activation]['val_loss'], label=activation)
plt.xlabel('Epochs')
plt.ylabel('Validation Loss')
plt.title('Validation Loss for Different Activation Functions')

plt.subplot(1, 3, 3)
for activation in activations:
    plt.plot(loss_history[activation]['test_loss'], label=activation)
plt.xlabel('Epochs')
plt.ylabel('Test Loss')
plt.title('Test Loss for Different Activation Functions')

plt.tight_layout()
plt.legend()
plt.show()
```



We notice that Identity and Tanh both perform well on the validation as well as test dataset. Identity is a linear activation function and Tanh is a non-linear activation function. Loss in case of Identity decreases faster then Tanh. Logistic activation function performs the worst out of all four. Relu also manages to decrease its loss almost as much as Identity and Tanh but it is slower than both.

In [19]:
```
plt.figure(figsize=(14, 6))
plt.subplot(1, 3, 1)
for activation in activations:
    plt.plot(accuracies[activation]['train_accuracy'], label=activation)
plt.xlabel('Epochs')
plt.ylabel('Training Accuracy')
plt.title('Training Accuracy for Different Activation Functions')

plt.subplot(1, 3, 2)
for activation in activations:
    plt.plot(accuracies[activation]['val_accuracy'], label=activation)
plt.xlabel('Epochs')
plt.ylabel('Validation Accuracy')
plt.title('Validation Accuracy for Different Activation Functions')

plt.subplot(1, 3, 3)
for activation in activations:
    plt.plot(accuracies[activation]['test_accuracy'], label=activation)
plt.xlabel('Epochs')
plt.ylabel('Test Accuracy')
plt.title('Test Accuracy for Different Activation Functions')

plt.tight_layout()
plt.legend()
plt.show()
```



The accuracy curves also show that while logistic fails to achieve good accuracy on train, val or test data, Identity, Tanh and Relu all perform well on all three datasets.

Grid Search

In [30]:
```
param_grid = {
    'solver': ['adam', 'sgd'],
    'learning_rate_init': [2e-5, 1e-4, 1e-3],
    'batch_size': [32, 64, 128]
}

grid_search = GridSearchCV(MLPClassifier(hidden_layer_sizes=(128, 64, 32), max_i
                           param_grid, cv=5, scoring='accuracy', verbose=2, n_jo
grid_search.fit(X_train, y_train)

best_params = grid_search.best_params_
print("Best parameters found: ", best_params)

results_df = pd.DataFrame(grid_search.cv_results_)
print("Full grid search results:")
print(results_df[['params', 'mean_test_score', 'std_test_score', 'rank_test_scor
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
Best parameters found:  {'batch_size': 128, 'learning_rate_init': 0.0001, 'solve
r': 'adam'}
Full grid search results:
                                               params  mean_test_score  \
0   {'batch_size': 32, 'learning_rate_init': 2e-05...         0.807375
1   {'batch_size': 32, 'learning_rate_init': 2e-05...         0.761375
2   {'batch_size': 32, 'learning_rate_init': 0.000...         0.804625
3   {'batch_size': 32, 'learning_rate_init': 0.000...         0.803125
4   {'batch_size': 32, 'learning_rate_init': 0.001...         0.612625
5   {'batch_size': 32, 'learning_rate_init': 0.001...         0.480375
6   {'batch_size': 64, 'learning_rate_init': 2e-05...         0.792625
7   {'batch_size': 64, 'learning_rate_init': 2e-05...         0.706250
8   {'batch_size': 64, 'learning_rate_init': 0.000...         0.813125
9   {'batch_size': 64, 'learning_rate_init': 0.000...         0.799375
10  {'batch_size': 64, 'learning_rate_init': 0.001...         0.656500
11  {'batch_size': 64, 'learning_rate_init': 0.001...         0.680750
12  {'batch_size': 128, 'learning_rate_init': 2e-0...         0.776375
13  {'batch_size': 128, 'learning_rate_init': 2e-0...         0.641875
14  {'batch_size': 128, 'learning_rate_init': 0.00...         0.814875
15  {'batch_size': 128, 'learning_rate_init': 0.00...         0.772625
16  {'batch_size': 128, 'learning_rate_init': 0.00...         0.693000
17  {'batch_size': 128, 'learning_rate_init': 0.00...         0.741750

    std_test_score  rank_test_score
0         0.004667                3
1         0.005868               10
2         0.012934                4
3         0.011799                5
4         0.017750               17
5         0.069929               18
6         0.005637                7
7         0.012009               12
8         0.005078                2
9         0.007935                6
10        0.028175               15
11        0.013581               14
12        0.007109                8
13        0.019780               16
14        0.003980                1
15        0.007282                9
16        0.020959               13
17        0.013927               11
C:\Users\Ritika\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\neural_network
\_multilayer_perceptron.py:690: ConvergenceWarning: Stochastic Optimizer: Maximum
iterations (100) reached and the optimization hasn't converged yet.
  warnings.warn(
```

We performed grid search on the following parameters:

- 'solver': ['adam', 'sgd']
- 'learning_rate_init': [2e-5, 1e-4, 1e-3]
- 'batch_size': [32, 64, 128]

and found that the best hyperparameters are:
{'batch_size': 128, 'learning_rate_init': 0.0001, 'solver': 'adam'}

MLP Regressor

Using "Relu" and "identity" Activation Function

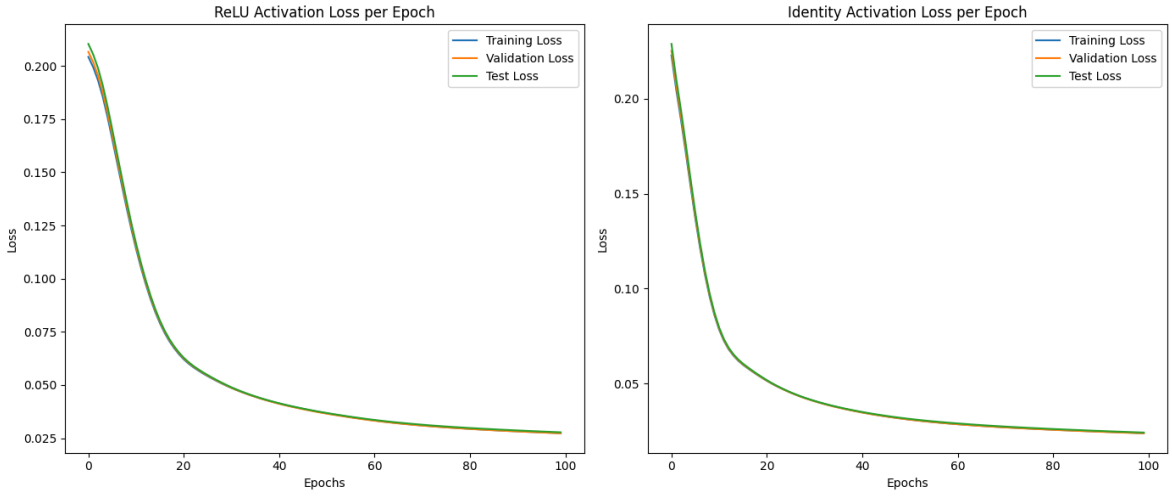
Here we are basically doing dimensionality reduction due to the way in which our layer sizes are designed. We are using the "relu" activation function and "identity" activation function. We are trying to regenerate the images using the MLP regressor (here the value of a = 32 but once it will cross 100 the accuracy of regeneration will saturate).

In [ ]:
```
def get_feature_vectors(model, data, layer_index=2):
    layer_output = data
    for i, layer_weights in enumerate(model.coefs_):
        layer_output = safe_sparse_dot(layer_output, layer_weights) + model.inte
        if i == layer_index:
            if model.activation == 'relu':
                layer_output = np.maximum(0, layer_output)
            elif model.activation == 'identity':
                pass
            break
    return layer_output
```

In [ ]:
```
layer_sizes = [128, 64, 32, 64, 128]
activations = ['relu', 'identity']
generated_images = {}
loss_history = {activation: {'train_loss': [], 'val_loss': [], 'test_loss': []}
feature_vectors = {activation: {'train': [], 'val': [], 'test': []} for activati
accuracies = {activation: {'train_accuracy': [], 'val_accuracy': [], 'test_accur

for activation in activations:
    mlp_regr = MLPRegressor(hidden_layer_sizes=layer_sizes, activation=activatio
                            solver='adam', max_iter=1, learning_rate_init=2e-5,
    for epoch in range(100):
        mlp_regr.fit(X_train_split, X_train_split)

        # Calculate training loss
        X_train_pred = mlp_regr.predict(X_train_split)
        train_loss = np.mean((X_train_split - X_train_pred) ** 2)
        loss_history[activation]['train_loss'].append(train_loss)

        # Calculate validation loss
        X_val_pred = mlp_regr.predict(X_val_split)
        val_loss = np.mean((X_val_split - X_val_pred) ** 2)
        loss_history[activation]['val_loss'].append(val_loss)

        # Calculate test loss
        X_test_pred = mlp_regr.predict(X_test)
        test_loss = np.mean((X_test - X_test_pred) ** 2)
        loss_history[activation]['test_loss'].append(test_loss)

        # Calculate training accuracy
        training_accuracy = accuracy_score(y_train_split, y_train_pred)
        accuracies[activation]['train_accuracy'].append(training_accuracy)

        # Calculate validation accuracy
        validation_accuracy = accuracy_score(y_val_split, y_val_pred)
```

```
        accuracies[activation]['val_accuracy'].append(validation_accuracy)

        # Calculate test accuracy
        test_accuracy = accuracy_score(y_test, y_test_pred)
        accuracies[activation]['test_accuracy'].append(test_accuracy)

    # Store feature vectors for training and test data
    feature_vectors[activation]['train'] = get_feature_vectors(mlp_regr, X_train
    feature_vectors[activation]['val'] = get_feature_vectors(mlp_regr, X_val_spl
    feature_vectors[activation]['test'] = get_feature_vectors(mlp_regr, X_test)



    # Generate images
    generated_images[activation] = mlp_regr.predict(X_test[:10])
```

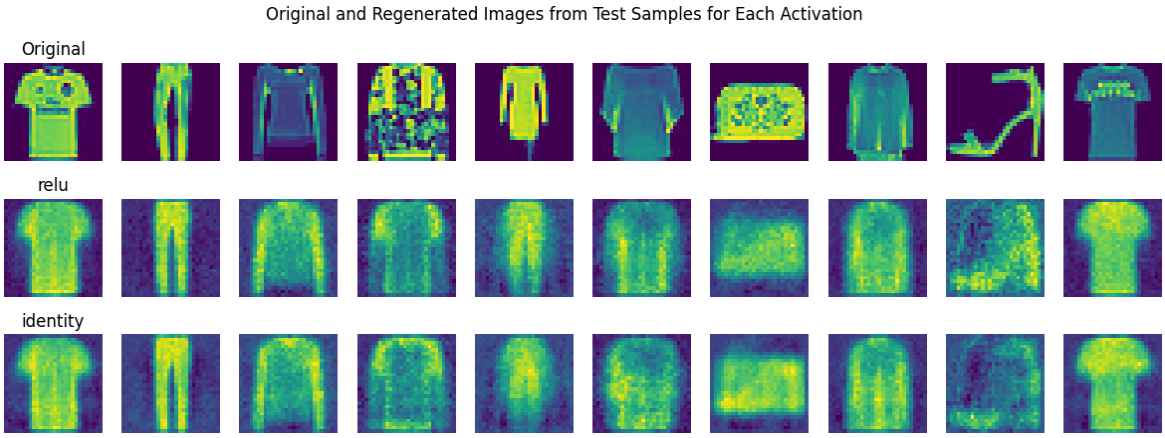
In [28]:
```
# Create a 1x2 grid of subplots for relu and identity activation losses
fig, axs = plt.subplots(1, 2, figsize=(14, 6))

# Plot losses for the 'relu' activation function
axs[0].plot(loss_history['relu']['train_loss'], label='Training Loss')
axs[0].plot(loss_history['relu']['val_loss'], label='Validation Loss')
axs[0].plot(loss_history['relu']['test_loss'], label='Test Loss')
axs[0].set_title('ReLU Activation Loss per Epoch')
axs[0].set_xlabel('Epochs')
axs[0].set_ylabel('Loss')
axs[0].legend()

# Plot losses for the 'identity' activation function
axs[1].plot(loss_history['identity']['train_loss'], label='Training Loss')
axs[1].plot(loss_history['identity']['val_loss'], label='Validation Loss')
axs[1].plot(loss_history['identity']['test_loss'], label='Test Loss')
axs[1].set_title('Identity Activation Loss per Epoch')
axs[1].set_xlabel('Epochs')
axs[1].set_ylabel('Loss')
axs[1].legend()

# Adjust layout for better visualization
plt.tight_layout()
plt.show()
```



Identity does a better job at reducing loss, however Relu also performs almost as good.

Plotting regenerated images

In [29]:
```
fig, axes = plt.subplots(3, 10, figsize=(15, 5))

for j in range(10):
    axes[0, j].imshow(X_test[j].reshape(28, 28))
    axes[0, j].axis('off')
    if j == 0:
        axes[0, j].set_title('Original')

for i, activation in enumerate(activations):
    for j in range(10):
        axes[i + 1, j].imshow(generated_images[activation][j].reshape(28, 28))
        axes[i + 1, j].axis('off')
        if j == 0:
            axes[i + 1, j].set_title(activation)
plt.suptitle('Original and Regenerated Images from Test Samples for Each Activat
plt.show()
```



MLP Classifiers on new set of extracted features

In [31]:
```
# Define new MLPClassifier model with 2 hidden layers of size 'a' = 32
new_relu = MLPClassifier(hidden_layer_sizes=(32, 32), max_iter=200,
                                    solver='adam', learning_rate_init=2e-5)
new_identity = MLPClassifier(hidden_layer_sizes=(32, 32), max_iter=200,
                                        solver='adam', learning_rate_init=2e-5)

# Fit these new classifiers on the training feature vectors
feature_vectors_relu_train = feature_vectors['relu']['train'].reshape(-1, 32)
feature_vectors_identity_train = feature_vectors['identity']['train'].reshape(-1

new_relu.fit(feature_vectors_relu_train, y_train_split)
new_identity.fit(feature_vectors_identity_train, y_train_split)

# Predict and evaluate accuracy on val and test feature vectors
feature_vectors_relu_val = feature_vectors['relu']['val'].reshape(-1, 32)
feature_vectors_identity_val = feature_vectors['identity']['val'].reshape(-1, 32

y_pred_relu_val = new_relu.predict(feature_vectors_relu_val)
y_pred_identity_val = new_identity.predict(feature_vectors_identity_val)

feature_vectors_relu_test = feature_vectors['relu']['test'].reshape(-1, 32)
```

```
feature_vectors_identity_test = feature_vectors['identity']['test'].reshape(-1,

y_pred_relu = new_relu.predict(feature_vectors_relu_test)
y_pred_identity = new_identity.predict(feature_vectors_identity_test)

accuracy_relu_train = accuracy_score(y_train_split, new_relu.predict(feature_vec
accuracy_identity_train = accuracy_score(y_train_split, new_identity.predict(fea

accuracy_relu_val = accuracy_score(y_val_split, y_pred_relu_val)
accuracy_identity_val = accuracy_score(y_val_split, y_pred_identity_val)

accuracy_relu = accuracy_score(y_test, y_pred_relu)
accuracy_identity = accuracy_score(y_test, y_pred_identity)

print("Accuracy for new classifier (ReLU) on training set:", accuracy_relu_train
print("Accuracy for new classifier (Identity) on training set:", accuracy_identi

print("Accuracy for new classifier (ReLU) on validation set:", accuracy_relu_val
print("Accuracy for new classifier (Identity) on validation set:", accuracy_iden

print("Accuracy for new classifier (ReLU):", accuracy_relu * 100, "%")
print("Accuracy for new classifier (Identity):", accuracy_identity * 100, "%")
```

```
C:\Users\Ritika\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\neural_network
\_multilayer_perceptron.py:690: ConvergenceWarning: Stochastic Optimizer: Maximum
iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
Accuracy for new classifier (ReLU) on training set: 64.15625 %
Accuracy for new classifier (Identity) on training set: 71.671875 %
Accuracy for new classifier (ReLU) on validation set: 63.24999999999999 %
Accuracy for new classifier (Identity) on validation set: 71.5625 %
Accuracy for new classifier (ReLU): 62.55 %
Accuracy for new classifier (Identity): 70.5 %
C:\Users\Ritika\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.11_qbz5n
2kfra8p0\LocalCache\local-packages\Python311\site-packages\sklearn\neural_network
\_multilayer_perceptron.py:690: ConvergenceWarning: Stochastic Optimizer: Maximum
iterations (200) reached and the optimization hasn't converged yet.
  warnings.warn(
```

Even though we reduced the number of features from 784 to 32, the accuracy of the MLP classifier is still decent. This is because the features that we have extracted are dense representation that contain high level information about the images which our regressor has learned. Thus even though we have reduced the complexity, the most important information is still retained.

In [ ]: